Declarative Ajax and Client Side Evaluation of Workflows using iTasks

Rinus Plasmeijer
Radboud University Nijmegen, The Netherlands
rinus@cs.ru.nl

Jan Martin Jansen
Netherlands Defence Academy, Faculty of Military Sciences, Den Helder, The Netherlands
jm.jansen.04@nlda.nl

Pieter Koopman
Radboud University Nijmegen, The Netherlands
pieter@cs.ru.nl

Peter Achten
Radboud University Nijmegen, The Netherlands
P.Achten@cs.ru.nl

Abstract

Workflow systems coordinate tasks of humans and computers. The iTask system is a recently developed toolkit with which workflows can be defined declaratively on a very high level of abstraction. It offers functionality which cannot be found in commercial workflow systems: workflows are constructed dynamically depending on the outcome of earlier work, workflows are strongly typed, and they can be of higher order. From the specification, a web-based multi-user workflow system is generated. Up until now we could only generate thin clients. All information produced by a worker triggers a round trip to the server. For real world workflows this is unsatisfactory. Modern Ajax web technology to update part of a web page is required, as well as the ability to execute tasks on clients. The architecture of any system that supports such features is complex: it manages distributed computing on clients and server which generally involves the collaboration of applications written in different programming languages. The contribution of this paper is that we integrate partial updates of web pages and client side task evaluation within the iTask system, while retaining its approach of a single language and declarative nature. The workflow designer uses light-weight annotations to control the run-time behavior of work. The iTask implementation takes care of all the hard work under the hood. Arbitrary tasks (functional programs) can be evaluated at web clients. When such a task cannot be evaluated on the client for some reason, the system switches to server side evaluation. All communication and synchronization issues are handled by the extended iTask system.

Categories and Subject Descriptors D.1.1 [*Applicative (Functional) Programming*]; D.3.2 [*Language Classifications*]: Applicative (functional) languages; H.4.1 [*Office Automation*]: Workflow management

General Terms Algorithms, Design

PPDP'08, July 15–17, 2008, Valencia, Spain.

Keywords Workflow Systems, Web Programming, Clean, Generic Programming, iData, iTask, Sapl, Ajax

1. Introduction

A workflow system is a computer system that coordinates the work that has to be done by *human workers* in collaboration with *computers*. Workflow systems are challenging real-world applications because they need to handle many things. First of all, a workflow system has to provide a way to specify workflows: *what* are the *tasks* that have to be done, how do these tasks *depend* on each other, and *who* should do them? The specification is used by the system at run-time for the real-time coordination and monitoring of the actual work being performed. Hence, somehow a mapping has to be made between the workflow specification and the real work that has to be done given the concrete human and software resources which are available. Daily work can be structured in quite a complex way which has direct consequences for the way tasks are depending on each other. The result of the work of one worker might determine the work of many others in both a positive or negative way. One needs a good understanding of how tasks depend on each other, and one also needs a sufficiently powerful specification formalism to express such complicated dependencies. In addition, one has to control a process which is quite dynamic: the amount and kind of work, the time it takes to do a job (ranging from split seconds to months), the number of available workers, the allocation of resources (both human, software, and hardware), they may all vary over time and may depend on the concrete work that takes place. Last but not least, one generally has to deal with a technically complicated distributed, heterogeneous environment: people working together all over the world using their own personal computer, pda's, mobile phone, and so on.

How to express this all? How to control this given a specification? It should be clear that a software system that can deal with all the above is bound to be complex. There exist many, mainly commercial, workflow systems. Examples are Business Process Manager, COSA Workflow, FLOWer, i-Flow 6.0, Staffware, Websphere MQ Workflow, and YAWL. Although these systems all have their own way of dealing with the challenges mentioned above, they also have a lot in common. Usually the systems are based on Petri-nets. The advantage is that dependencies between tasks can be depicted which makes them attractive to non-experts, while these drawings can straightforwardly be mapped to a corresponding Petri-net. This

Petri-net is used at run-time as scheme to control the real work to do. Furthermore the net can be used as a formal model at compile-time to determine desired properties of the specified workflow: one can calculate reachability of a certain task or determine the absence of deadlock. The kind of task dependencies one can specify in these systems, the so-called *workflow patterns*, are summarized in [14], together with a discussion of the systems mentioned above. However, the use of Petri-nets as semantic model also has big disadvantages. The nets are rather static and only first order: tasks cannot deliver new tasks to do. Hence they cannot be used to describe the dynamic way of working that takes place in the real world and therefore they can only be used for the specification of static tasks.

The main research question that we address in this paper has been asked to us by industry being confronted with the limitations of the current systems: can declarative programming, and functional programming in particular, provide new concepts and implementation methods and tools for workflow systems that can deal with the dynamic behavior of daily work? In answering this question, we have developed the iTask toolkit [12] as a first step towards a realistic workflow system. This toolkit is a web-based combinator library written in the lazy, purely functional programming language Clean. The novel and declarative contributions that this toolkit provides, which cannot be found in the existing commercial systems, are:

- workflows are constructed fully dynamically instead of statically: they can depend on the intermediate inputs and outputs that are yielded by workers and computations;
- workflows can be higher-order, i.e. yield partially evaluated tasks which can be passed around for further evaluation to other workers at other locations;
- workflow cases are specified as pure, strongly-typed functional expressions, using the predefined iTask combinators;
- the workflow application can handle multiple workers, multiple tasks, and multiple clients dynamically, yet everything is controlled by one, *single* Clean application running on the server;
- the specification of the workflow is *executable*; all implementation details like web-page generation, web-page handling, client-server communication and database storage handling is handled fully automatically by making intensive use of generic programming techniques [8, 3]: from the *types* being used the required *code* is *generated* fully automatically.

Tasks have to be offered to the workers in such a way that it is clear what they have to do. The iTask application generates, given the workflow specification and the work that has been done so far, an appropriate web page for each user. The key advantage of using browsers to display the work to do, is of course that they are available on any thinkable platform. No special software needs to be installed to connect iTask workflow users. Workflow systems are distributed software systems, hence it makes sense to not only deploy web technology for rendering purposes, but also for the distribution, communication, and control of tasks. However, although the web seems to be very suited for all this, it is actually technically quite difficult to realize the rendering and communication automatically from a given declarative workflow specification. Workflow systems exhibit *state*, support *multiple users*, and guide the *flow of work*. Neither of these concepts are readily supported by the web and hence additional software is needed for the realization. Commonly, a web application which guides a user through several working steps does not consist of one, single application. The implementation often consists of a collection of software applications and scripts, written in several languages, which somehow together do the job: one can think of HTML-code, php-scripts, Ajax-scripts, SQL-queries. Since they are commonly not generated from one single source code, it is very hard to design, implement and maintain systems which such an architecture. In the iTask system all software *is* generated from one single source in Clean. To understand what the application is doing, one only needs to look at the iTask specification. It is a specification on a very high level of abstraction which can be read as if we are dealing with an ordinary simple desktop application. We take full advantage of the fact that we are working with a pure functional language. First of all we solve the lack-of-state problem of the web, by using generic programming techniques to store the state of the interactive elements, the iTasks, *only*. Because we have the most recent states of the iTasks at our disposal, we only need to rerun the function that represents the program and provide it with the most recent input action of any worker to advance to the next state. This reduces the programming burden on the workflow developer. It allows her to focus on the workflow case, rather than its implementation. Another advantage of such an approach is that one obtains a clean separation between the workflow *specification* and its *implementation*.

In this paper we explain the use and implementation of two new important features added to the iTask toolkit. In the old system, any event received from an iTask user is handled by the single iTask application on the server. It computes the next state and calculates a whole new web page for a particular user showing her the new tasks to do. New web technology such as Ajax [6], makes it possible to update only a part of a page. Updating only the relevant part of a page improves the behavior of the web application in a way that resembles desktop behavior. The first feature is that we incorporate partial page updates. This is a challenge since the iTask system dynamically calculates which part of the page has to be updated because it depends on the state of the task being performed and the state of the work of all other users. In most existing systems the part of the page to be updated is fixed rather than computed dynamically. The second feature is that we want that the program executing the tasks can run partially on the client instead of on the server. Client side evaluation is essential to eliminate delays associated with the communication between server and client. The impact of this feature can not be overestimated because it is fundamental to create coarse grained computational tasks on clients with rich interaction and quick response times (think of modern day web applications like Google Docs and gmail). There are three ways to obtain client side evaluation of tasks in a browser: plugins, JavaScript, or Java code. The disadvantages of plugins is the explicit installation that is required. In the current iTasks system we use Java since it seems better suited for the large applications that have to be run on the client than JavaScript.

Instead of a single server, one can also think of using several servers, as well as tasks that are migrating over the internet. Distribution of client tasks is also required when one wants to work with *distributed document repositories* that can be accessed by client workflows. This is not addressed in this paper, but will be subject of future research. The feature of client side workflows also challenges the underlying architecture of the iTask toolkit.

In this paper we show how these two major web techniques can nevertheless be incorporated smoothly within the iTask toolkit, while fully retaining its declarative nature:

- We rearrange the iTask toolkit in such a way that worker-tasks automatically use the asynchronous, partial page update technology that is offered by Ajax. Besides this default arrangement, we allow the workflow designer to *annotate* workflow expressions in a light-weight way to give fine-grained control of other parts of the workflow application.
- We rearrange the iTask toolkit in such a way that *arbitrary workflow task expressions* can be evaluated at the client side.

For the workflow designer this is only a matter of a simple annotation in the existing specification.

The workflow engineer can use the new contributions of the iTask toolkit by annotating ordinary iTask applications. The implementation of these new annotations is however rather challenging. We wish to evaluate complete task expressions on the client instead of the server which requires that we can evaluate full Clean code on the client side within the browser. Worker actions can have non-local effects, hence we need to implement some sort of *synchronization*. We show that this can be implemented without loss of the semantic model of the system without client side evaluation and partial page updates.

The declarative nature of the iTask toolkit is retained by implementing an *evaluation strategy* that can automatically switch between client side evaluation and server side task rewriting if necessary. The details are presented in Sections 5 and 6, but roughly speaking this means that the system can perform tasks on the client side (within a browser) as well as on the server side (within the server application). Moreover, if client side evaluation is no longer possible (because of a non-local effect of a remote worker, or because the local computation requires a server resource), the system automatically can continue to perform the computation on the server side. The workflow designer does not have to specify this, unlike other approaches as for instance in Hop [13, 10] (see also Sect. 7). This implies that the approach as described in this paper is not only more declarative, but also more robust: it can handle situations dynamically that would otherwise be considered programming errors.

The iTask toolkit has been created in Clean. A concise overview of the syntactical differences with Haskell is [2]. We assume the reader is familiar with the concept of generic programming.

We start with a short overview of the iTask combinator system in Sect. 2. The new annotations are introduced in Sect. 3. Their ease of usage contrasts strongly with their implementation. To understand why, we present the basic architecture of the standard implementation in Sect. 4. The high level specification of workflows offered by the iTask system is achieved due to the fact that the system is able to reconstruct the state of evaluation of all tasks of all users, the so called *Task Tree*. To avoid the *Task Tree* from growing infinitely, a task expression is rewritten by its result in a similar way as function applications are rewritten by their result. This is called *Global Task Tree Rewriting*. In Sect. 5 we discuss the implementation consequences of asynchronous partial page updates and introduce *Local Task Tree Rewriting*. In Sect. 6 we do the same for client side evaluation and introduce *Client Side Local Task Tree Rewriting*. Related work is presented in Sect. 7 and we conclude in Sect. 8.

2. Introduction to iTasks

In this section we give a concise overview of the iTask system. First we select the combinators that are used in this paper (Sect. 2.1). We present the complete code of a small, but representative case study (Sect. 2.2). Finally, we discuss opportunities for optimization (Sect. 2.3).

2.1 The iTask Combinators

Although the iTask system supports all common workflow patterns found in commercial workflow systems ([14] gives an excellent overview), it is beyond the scope of this paper to discuss them all. The selection of iTask combinators that we use in this paper are shown in Fig. 1.

In the iTask toolkit tasks are represented by the opaque type (`Task a`). The primitive task (`editTaskPred` a p) generates a web form for values that have the type of the initial value a. The predicate p is

```
:: Task        a
:: Pred        a := a → (Bool, [BodyTag])
:: LabeledTask a := (String,Task a)
:: UserId        := Int

editTaskPred   :: a (Pred a)                  →Task a     | iData a
editTask       :: a                           →Task a     | iData a
(=>>) infix  1 :: (Task a)    (a →Task b)     →Task b     | iData b
return_V       :: a                           →Task a     | iData a
buttonTask     :: String      (Task a)        →Task a     | iData a
chooseTask     :: HtmlCode [LabeledTask a]    →Task a     | iData a
(-||-) infixr 3 :: (Task a)        (Task a)   →Task a     | iData a
(-&&-)infixr 4 :: (Task a)        (Task b)    →Task (a,b) | iData a
                                                         & iData b
(?>>) infixr 5 :: HtmlCode        (Task a)    →Task a     | iData a
(@:)  infix  3 :: UserId   (LabeledTask a)    →Task a     | iData a
```

Figure 1. The selection of iTask toolkit combinators

used to impose further constraints on entered values (they at least have to be of correct type). Only when the worker has entered a value of correct type that also meets the given predicate the task can be finished by the worker and that value is returned. If the predicate p is not needed one can use `editTask` a. The type class restriction `| iData a` at the end of the type signature guarantees that this function works for *any* type `a` provided that all generic instances for this type of the generic functions being used are available. The compiler can automatically *derive* these instances on request of the programmer (see Sect. 2.2). An edit task for a string is specified as:

```
et :: Task String
et = editTask "Finished" "edit string here"
```

The initial string is `"edit string here"`. The task is finished when the user presses the button labeled `Finished`.

The iTask library uses the monadic combinators =>> and `return_V` for their standard purposes. The task `return_V "Approved"` is a task that returns the string `"Approved"` without any user interaction.

A `buttonTask` s t activates the task t after the user has pressed the button labeled by the string s. As an example: the task `yt` (`nt`) yields the string `"Approved"` (`"Rejected"`) when the user presses the button labeled `Yes` (`No`).

```
yt :: Task String
yt = buttonTask "Yes" (return_V "Approved")

nt :: Task String
nt = buttonTask "No"  (return_V "Rejected")
```

`chooseTask` *html* $[(l_0, t_0) \ldots (l_n, t_n)]$ allows the worker to preselect one labeled task t_i from the list. After the choice, the other tasks have disappeared. For example

```
ct = chooseTask [Txt "choose"]
                [("Yes", return_V "Approved")
                ,("No",  return_V "Rejected")
                ,("Edit",et)]
```

prompts the user with the text `choose` and offers three buttons labeled `Yes`, `No`, and `Edit`. After using one of the first two buttons `ct` will be finished and deliver the indicated string. If the user presses the `Edit` button the iTasks system offers the user the edit task `et`.

The expression t-||-u offers tasks t and u simultaneously. As soon as either one is finished first, t-||-u is also finished. Any work in the other task is discarded. The -||- combinator is very useful to express work that can be aborted by other workers or external circumstances. In

```
ot = yt -||- nt -||- et
```

the iTasks system offers the task `yt`, `nt`, and `et` simultaneously. Any edit work in `et` is discarded when the user presses one of the buttons labeled `Yes` or `No`. Discarding of work is prevented in `ct` where the user chooses the task to be done before she starts editing.

Tasks can be composed sequentially by the monadic $\gg\!=$ operator. For instance the string resulting from `ot` can be edited until the `Done` button is pressed by executing `ot` $\gg\!=$ `editTask "Done"`.

If one really needs both results of tasks t and u, then this is expressed by t `-&&-` u, which runs both tasks to completion and returns both results. For instance, if we need a string and an integer (with default value 5) we can use the task:

```
at :: Task (String, Int)
at = ot -&&- editTask "Done" 5
```

It is useful to provide the worker with additional information *info* while she is working on a task t. This is expressed with *info* `?>>`t. Finally, any task t labeled with l can be assigned to some user with user identification value i with i`@:`(l,t). We illustrate this in the next case study.

2.2 Case Study

The case study is a tiny *personnel administration* workflow (see also Fig. 2) in which two co-workers A and B need to perform the same task of administrating personnel information simultaneously. In both cases they can enter information, double check their input, and submit the information. If worker A is the first to finish, then the whole workflow case terminates. When worker B finishes first, the result is given to worker A, who can inspect and adjust this value. Note that at this stage, worker A now has the choice of either finishing her own version, or decide to continue work on B's result.

This case study is a representative example of a workflow situation. It has unpredictable execution times of tasks (as both workers can decide how much time to consume), non-locality (race between two workers and the outcome of worker B affects the tasks of worker A), and potential distribution of local work (both workers could perform the given task locally on the client).

Let `workflow` be the specification of the case study workflow. Every multi-user iTask program has the following preamble:

```
module admin                                                    1.
import StdEnv, StdiTasks                                        2.

Start world = multiUserTask [] workflow world                   3.
```

The main function is `Start` (line 3), which calls two wrapper functions: `doHtmlServer` connects the Clean application to a server in such a way that one can surf to the web-page with the name `admin`; `multiUserTask` generates a multi-user workflow infrastructure for the specification `workflow`.

In the case study worker A and B perform the same task, `person_admin`. We capture this pattern concisely by means of a parameterized workflow function `delegate` which is also parameterized with the worker identification values 0 (worker A) and 1 (worker B).

```
workflow = delegate 0 1 person_admin                            4.

delegate :: UserId UserId (a→Task a)→Task a | iData a            5.
delegate userA userB taskf = a-||-b                             6.
where                                                           7.
    a = userA @: ("Task of A",taskf createDefault)              8.
    b = userB @: ("Task of B",taskf createDefault               9.
        >>= λrb→userA @: ("A checks B",taskf rb))               10.
```

The function `delegate` specifies the main structure of the workflow as described above: two tasks (`a` and `b`) are created simultaneously (`-||-`). The first task is provided with an initial default value (`createDefault`), and this is the task that needs to be performed by worker A (line 8). The second task is provided with the same initial default value, and needs to be performed by worker B (line 9). When finished, worker B has produced `rb`, which is passed along via the monadic bind combinator $\gg\!=$ to worker A again, who can decide to work with `rb` (line 10).

The task `person_admin` that is performed by worker A and B double-checks filling in a personnel record of type `Person`:

```
:: Person = { name        :: String,   e_mail :: String         11.
            , dateOfBirth :: HtmlDate, gender :: Gender }       12.
:: Gender = Female | Male                                       13.

person_admin :: Person→Task Person                              14.
person_admin p = doubleCheck p checkPerson                      15.

checkPerson :: Pred Person                                      16.
checkPerson {name,e_mail}                                       17.
  | name == "" = (False, [Txt "Please fill in your name"])      18.
  | not ok     = (False, [Txt "Incorrect e-mail address"])      19.
  | otherwise  = (True,  [])                                    20.
where                                                           21.
  ok = not (isMember '@' (fromString e_mail)) || e_mail == ""   22.
```

The predicate `checkPerson` determines whether the worker did a good job. Double-checking a worker's output is also a parameterized workflow function:

```
doubleCheck :: a (Pred a)→Task a | iData a                      23.
doubleCheck a p                                                 24.
= [Txt "Please fill in the form:"]                              25.
  ?>> editTaskPred a p >>= λna→                                 26.
  chooseTask [ Txt "Received information:"                      27.
             , toHtml na, Txt "Is it correct?" ]                28.
             [ ("Yes", return_V na)                             29.
             , ("No", doubleCheck p na)]                        30.
```

(`doubleCheck a p`) uses (`editTaskPred a p`) (line 26) to generate a web form to enter a value of the type of `a`: again, the type class restriction guarantees that this is possible for the particular type of `a`. Once the worker has successfully entered a correct value, then this is passed monadically as `na` (line 26) to the next sub-task (lines 27-30): the value is displayed (`toHtml na`, line 28), and the same worker is asked to confirm whether she is sure about the information she has entered: if she confirms (line 29), then `doubleCheck` returns that value, and if she declines (line 30), then `doubleCheck` *recurses* with the new value.

What remains to be done is to include 'boilerplate' code for deriving instances of the custom data types of the required generic functions:

```
derive gForm   Person, Gender  // create form                   31.
derive gUpd    Person, Gender  // process edit operation        32.
derive gPrint  Person, Gender  // serialize value               33.
derive gParse  Person, Gender  // deserialize value             34.
```

This completes the case study.

2.3 Opportunities for optimization

The case study illustrates a number of opportunities for efficient evaluation: in the current iTask implementation, every worker action triggers a round-trip between the client browser and server application. The actions of worker A and B are largely independent: still, the application takes both current states into account whenever either worker submits information. This can be improved if the system would restrict itself only to the required information. Also, one can imagine that the complete `person_admin` task can be executed on the client, without any communication with the server. This requires on-client evaluation of arbitrary Clean code. In the next section, we present the novel extensions to the iTask system that allow the workflow engineer to specify these properties, while maintaining correct handling of multiple users and global effects.

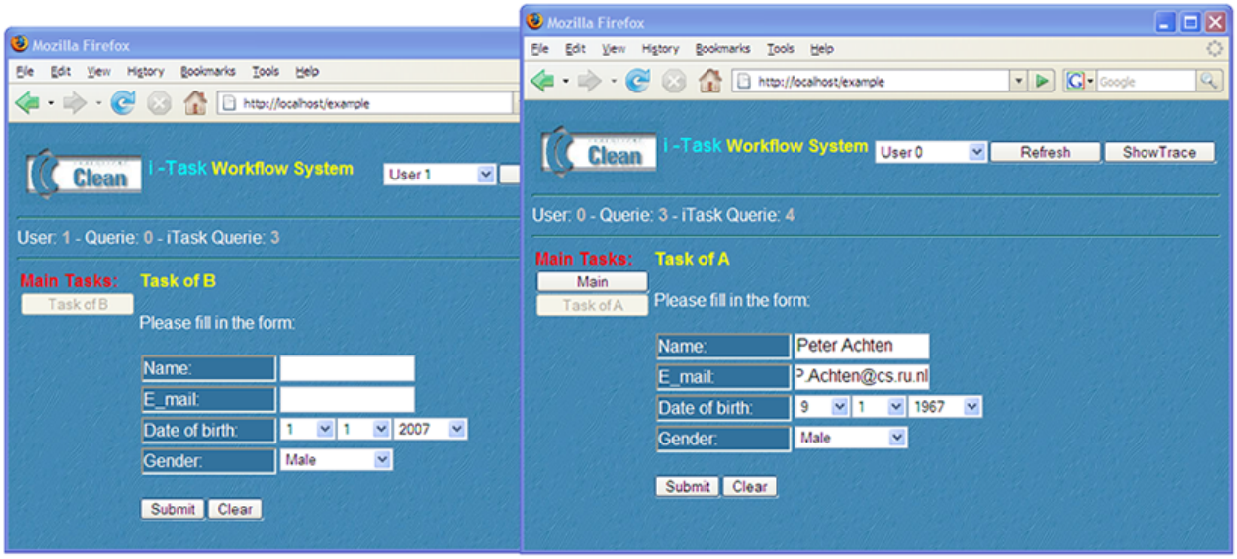

Figure 2. The case study with two workers: A (right window) and B (left window).

3. Controlling the evaluation of tasks

In this section we introduce two annotations that the workflow engineer can use to control the behavior of any task t. The annotations are (`UseAjax @>>` t) and (`OnClient @>>` t), and are implemented as type class instances:

```
:: SubPage = UseAjax | OnClient

class (@>>) infixl 7 b :: b (Task a) → Task a | iData a
instance @>> SubPage
```

3.1 The "UseAjax" annotation

Modern web browsers support Ajax-technology. Ajax allows web applications to define *call-back functions* on the client in JavaScript. When a client browser submits a request for a new page to the server it usually receives a complete new page and renders the new page. Using Ajax, the call-back function handles the response of the server instead of the browser. This happens asynchronously, hence the user can continue to work on the page in the browser while the request is being processed. With this technique web pages can be updated partially, which results in a much more responsive behavior that resembles desktop applications. As we will see, also the implementation can benefit from it: in many cases only the effect of the particular task being performed has to be calculated, instead of all tasks.

The workflow engineer can annotate any task expression. This requires some consideration, because Ajax imposes a performance penalty. As a rule of thumb, worker tasks (tasks assigned to a worker with the `@:` operator) are suitable candidates for annotation because they clearly form a unit of work and they own graphical estate on the web page. To support this rule of thumb, the workflow engineer can switch it on in the iTask library and hence readily create "Ajax threads" for explicit worker tasks. For some applications this default granularity might turn out to be too coarse grained. Using the `UseAjax` annotation allows the workflow engineer to create Ajax threads on any level, they may be invoked conditionally, they may be nested, and they may occur in recursive definitions.

The `UseAjax` facility is also a very useful feature because it can serve as an automatic backup mechanism when client site evaluation is somehow not possible (see hereafter).

3.2 The "OnClient" annotation

In Sect. 2.3 we suggested that the double checking personnel data task can be executed completely on a client instead of the server. The only change to the case study specification is adding the appropriate `OnClient` annotations in the `delegate` function:

```
delegate :: UserId UserId (a → Task a) → Task a | iData a          5.
delegate userA userB taskf = a -||- b                              6.
where                                                              7.
  a = userA@:("Task of A", OnClient @>> taskf createDefault)       8.
  b = userB@:("Task of B", OnClient @>> taskf createDefault        9.
      >>= λrb → userA@:("A checks B", OnClient @>> taskf rb))      10.
```

Any such annotated task is a "client thread", and is supposed to be executed on the client browser. Not every task can always be evaluated on the client. For instance, a task might inspect or change information in a database stored on the server side. Due to the non-locality of worker actions, their effect can only be determined with global knowledge of the state of worker tasks. This type of knowledge is only available on the server. Consequently, the `OnClient` annotation must be seen as a *wish*: if it is possible the task is evaluated on the client, but the evaluation strategy might be forced to do the work on the server. It is also possible that a client task is part of a larger task to be executed on the server. When the client task is finished one has to be able to switch back to the server for the continuation. Now we can appreciate the availability of the `UseAjax` annotation even more: whenever `OnClient` evaluation of a task is not possible we can simply change it into an Ajax call instead and calculate the task on the server.

3.3 Discussion

With the two new annotations, `UseAjax` and `OnClient`, the workflow engineer can control the evaluation of tasks in a light weight way. However, the implementation of these annotations is by no means light weight because it needs to handle many issues. One issue is that for every worker event it needs to figure out which Ajax thread (if any) has to handle the event. However, the event may cause the associated task to terminate. In that case the Ajax thread has to terminate as well, and the parent thread has to be activated to determine the next tasks to deal with. This can result in a cascade of activated-terminated Ajax threads. Another issue is that due to non-

locality of worker actions tasks may disappear, and consequently also their associated Ajax threads. In these cases the evaluation strategy has to resort to the standard evaluation technique. Switching of evaluation strategy is also vital for those `OnClient` tasks for which it turns out that they cannot be evaluated on the client. Using the Ajax infrastructure allows the iTask toolkit to turn a failing `OnClient` task automatically into a `UseAjax` task, and hence have the task calculated on the server. This can only be done if the server has either full knowledge of the states of all clients, or if it can completely reconstruct their state on demand. The iTask toolkit uses the latter strategy, and this is explained in Sect. 4. After that, we show how Ajax technology is incorporated in Sect. 5 and client side evaluation in Sect. 6.

4. Standard iTask Implementation

In order to appreciate the implementation of the new extensions to the iTask toolkit, we need to focus on its initial implementation. The material presented in this section is a revision of [12].

4.1 A Functional Approach

As discussed earlier, the initial iTask toolkit creates thin-client web applications. This means that the client browsers are used for rendering purposes only. All events of all web clients that correspond with the workers that are currently using the workflow application are sent to one single server application. This server iTask application is executed every time an event is received. The result is a new web page for the worker. This page depends only on the input event and the current state. The state is adapted by the iTask server application. A number of fundamental design decisions have been taken in the creation of the iTask toolkit. In a nutshell, these are:

1. There is a single declarative iTask specification such as the `admin` case study in Sect. 2 from which all code is generated.
2. Task editors have persistent state. A task editor displays the state and allows the user to alter this state in such a way that only values of the same type can be created. Rendering, editing, updating, and storing and retrieving values is all done generically.
3. An iTask program is a pure function, hence referentially transparant, and will produce the same result (and same effect) when applied to the same input (and state). More precisely, it will produce the same task editors in the same order.

An iTask application has an effect on its external world and keeps track of the various persistent storages. Its *state* is used for this purpose, and is discussed in Sect. 4.2. The evaluation of an iTask expression gives rise to the concept of a *Task Tree*, which is presented in Sect. 4.3. Finalized tasks are *rewritten* in an analogous way as graph reduction takes place in the implementation of functional languages. This is explained in Sect. 4.4.

4.2 The iTask State

To the workflow engineer, the task type (`Task a`) is opaque. Internally, it is a state transformer function of type:

```
:: Task a :== *TSt → (a,*TSt)
```

The state of an iTask application is the uniquely attributed `*TSt`. Every task is applied to a `*TSt` value, and returns a modified `*TSt` value, as well as the result of the work being performed, which is a value of type `a`. Although iTasks are programmed in a monadic style, it is the underlying uniqueness typing of Clean that guarantees that the `*TSt` value is passed single threadedly from one iTask transition function to another.

```
:: *TSt = { hst  :: *HSt,     activated :: Bool
          , html :: HtmlTree, params    :: TParams }
```

`*TSt` extends the uniquely typed iData state `*HSt` [11]. For this paper, two components of `*HSt` are relevant. The first is an accumulator in which the state of all web form editors are collected, such that they can be saved in persistent memory when the iTask application terminates. The second is the `*World` environment value that allows it to perform these operations effectively.

The boolean value `activated` acts as a *control token* passed from one combinator to another indicating which tasks have terminated, which tasks are active, and which tasks need to be activated. When a combinator is called, `activated` tells it whether it has to be activated or not. If its value is `False` the task will not be activated at all and a (default) value of proper type is returned immediately, generated by making use of the generic machinery. Otherwise the task is activated and the combinator is applied on the current `*TSt` state, possibly activating other iTask combinators in turn. When the combinator is returning a result, the corresponding task may or may not be terminated. If the task is not terminated, the returned `activated` value is `False` and a (default) value is again returned as result of the task. If the task is terminated, `activated` is set, and the value returned is the official result of the task. Since the task has ended, this result can be given to the next task or set of tasks which in turn are activated as well. Hence, a combinator always returns a result of proper type, but only the values returned from finished iTasks are meaningful and are passed to other activated tasks. The meaningless default values are only passed around.

The Html code generated by tasks is accumulated in the `html` field. The information for the intended worker is filtered out.

The remaining information is collected in the `TParams` record. This information is necessary to construct a `*TSt` value when needed.

```
:: TParams =  { userId      :: UserId
              , options     :: Options
              , taskNr      :: TaskNr }
:: TaskNr  :== [Int]
:: Options =  { tasklife    :: Lifespan
              , taskstorage :: StorageFormat
              , taskmode    :: Mode
              , gc          :: GarbageCollect }
```

The `userId` is the unique identification of the worker who has to perform the corresponding task. An iTask can have many options which are stored in the `options` field. For instance, the `Lifespan` option defines in which memory (in the web page on the client side, or on the server side in a relational database or in a file on disk) the status of the task is stored when the application ends. Last but not least, every iTask obtains a unique identification, for which the `tasknr` field in the `*TSt` state is used. Such a unique identifier is crucial in order to retrieve the iTask state information from the different persistent stores. Tasks are numbered dynamically, in the same way as chapters, sections and subsections are numbered in a book or in this paper: tasks on the same level are numbered subsequently, whereas a subtask `j` of task `i` is numbered `i.j`. Task numbering allows us to determine how tasks are related to each other. Just by looking at the task numbers we can figure out the ancestors of a task and which subtasks it has spawned. In the standard iTask implementation this knowledge is used for garbage collection of subtasks. We can now use it conveniently for our new annotations to determine which (parent) thread to activate when an event has occurred.

4.3 The Task Tree

An iTask application remembers its point of evaluation. In a language like C or Java the point of evaluation is remembered by using a stack. For iTasks it is better to use a tree, the so-called *Task*

Tree. The reason is that we are dealing with a multi user system: people can work on many tasks simultaneously. As a matter of fact, also one user can have several tasks she can work on at the same time. At any time we have to be able to administrate the progress made on any task by any worker. Furthermore we have seen that new tasks can be created while other existing tasks might not be needed anymore.

A tree structure is well suited for the administration of all this. iTasks depend on other iTasks and finally on basic iTask editors. The dependencies are determined completely by specified iTask combinators. The used combinators form the nodes in the Task Tree, the basic editors the leaves.

The contents of a Task Tree varies over time. An activated task might be changed into a finished task, new tasks can appear and complete old sub trees may be pruned because the corresponding tasks are no longer needed.

Fig. 3 depicts a snapshot of the Task Tree of the `admin` case study. User A (id 0) and user B (id 1) are working on their `person_admin` task. The user id (in the left upper corner), the iTask combinator name, and the task number are displayed in each node. User 0 finished an `editTaskPred` and is now working on a `chooseTask`. User 1 is still working on an `editTaskPred`. There are two threads created, one for each `person_admin` task. The grey area indicates which combinators belong to which thread.

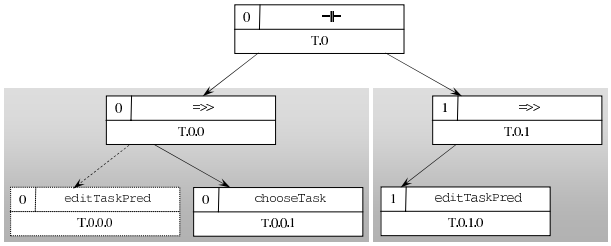

Figure 3. The Task Tree at work with the `admin` case study.

Although an iTask application can remember its previous point of evaluation, it is not realized by interrupting a running Clean application, waiting for the next event received from some user, and continue execution as one would do in a C implementation. The reason is that there is not one point of execution, there are several: all active worker tasks. So, a parallel evaluation order of the iTask specification would be appropriate, which is quite different from the normal order evaluation used in Clean. Implementing a parallel evaluation strategy is challenging and time-consuming. It turns out that there is an elegant, much simpler technique that achieves the same result. We exploit the fact that we are working with a *pure* functional language: the result of a function only depends on its arguments. We also make use of the iData library: every editor ever being used automatically stores its state in its specified (persistent) memory and this state is automatically recovered when the editor is activated again (see [11]). Reconstruction of the previous point of evaluation is accomplished by re-evaluation of the program with the new input received where at the same time the effect of old inputs is recovered automatically thanks to the iData being used.

Consequently, the Task Tree does not really exist: part of it is reconstructed via the re-evaluation of iTask combinators (they are just plain Clean functions), part of it is reconstructed using the stored information of the iData editors. In this way the Task Tree is reconstructed from scratch each time an event is received. This technique is also very suited for dealing in a robust way with a complicated hard to control distributed environment such as the internet.

On demand, the iTask application can show the contents of this virtual Task Tree to the user. In this way one can get an overview of who is doing what, which tasks are finished and what their results are.

4.4 Global Task Rewriting

Mature workflow systems should run for years and are used by many workers. This implies that the Task Tree as described above would grow indefinitely over time. For a real world workflow application this is of course unacceptable. The evaluation of an iTask is therefore optimized in a similar way as a function application is optimized in any implementation of a functional language: when a function has been evaluated, the function call is replaced by its result. Similarly, when a task is finished, it is replaced by its result. This is noticeable in the Task Tree as well: a combinator node in the Task Tree is replaced by the resulting task value. This *Global Task Rewriting* increases efficiency because Task Trees can be reconstructed much faster. Although not discussed in this paper, the iTask toolkit has iterative combinators such as `foreverTask` that repeat tasks infinitely many times. These can restart from scratch and even reuse the task numbers. In this way both the Task Tree and the task numbers have proper upper bounds.

The downside is that the implementation becomes more complicated as well. The iTask information stored in the persistent memories needs to be garbage collected which is not always trivial. As we will see, Task Tree rewriting also has an impact on the implementation of our new annotations.

5. Implementing Ajax calls via Local Task Rewriting

In this section we show how Ajax-threads are incorporated within the iTask toolkit. The key idea of Ajax is to enable JavaScripts that reside in a web page to engage in asynchronous communication with servers, a functionality that was strictly reserved to browsers before Ajax came along. The result of this technology is that one can create web pages that are constructed out of arbitrarily many 'classic' components (i.e. go along with the browser-server communication cycle) as well as arbitrarily many components that handle their private content with servers of their choice. In order to set up the asynchronous communication, a JavaScript creates a so-called *XML http request* object. With that object, it can communicate with any server of its choice. Usually, this communication is asynchronous, but synchronous communication is also possible. In case of asynchronous communication, a *callback function* is associated with the *XML http request* object, by overriding its *onreadystatechange* method. This function is called whenever the server has responded, and it will find that result in the *responseXML* data member of the *XML http request* object.

If we want to use this technology within the iTask framework we will need to create and store the proper callback functions. These callback functions will necessarily update the Task Tree *locally* instead of globally as described in Sect. 4. We make use of the property that a Task Tree cannot only be reconstructed from the root of the tree: any sub tree can be reconstructed in a similar way as well. The reason is that due to referential transparancy, the same Task Tree will be reconstructed, and this property also holds for any sub tree. So, we can reconstruct and rewrite the Task Tree *locally*, i.e. starting from any node in the tree if only we can store and determine the callback function that handles this part of the tree. This is discussed in Sect. 5.1. Due to possible non-local effects of worker tasks, we may need to switch between local Task Tree rewriting and global Task Tree rewriting. This is described in Sect. 5.2. Finally, we discuss what has been achieved after this step in Sect. 5.3.

5.1 Thread Storage and Creation

Every sub tree of the Task Tree has been created by one of the iTask combinators, a fragment of which has been displayed in Fig. 1. To reconstruct a sub tree we have to know which iTask combinator (thread) is responsible for its construction and we need to know with which arguments this function has been called. This information has to be stored somewhere such that we can re-evaluate the function later, as a special kind of callback function. So, we must be able to store and retrieve functions. Clean already has powerful means for doing that. By using Dynamics, any type, including function types, can type safely be stored and even be exchanged between independently programmed Clean applications [15]. Exchange of dynamics between two applications requires the presence of a dynamic linker. Loading dynamic code and data with a linker consumes a significant amount of time. Because we are dealing with one and the same server application, this is not necessary. We only make use of Clean's ability to serialize and de-serialize functions. The two functions that do this are `serializeClean` and `deserializeClean`:

```
serializeClean   :: (Task a) → CleanSerialization
deserializeClean :: CleanSerialization → Task a

:: CleanSerialization :== String
```

Note that type correctness is no longer automatically guaranteed, so our storage administration should better be correct, which is assured by the way they are created and used.

```
:: ThreadTable :== [TaskThread]
:: TaskThread  =  { thrCallback :: CleanSerialization
                  , thrParams   :: TParams }
insertNewThread   :: TaskThread *TSt → *TSt
deleteThreads     :: TaskNr     *TSt → *TSt
findThreadInTable :: TaskNr     *TSt → (Maybe TaskThread,*TSt)
```

In the `ThreadTable` threads are stored in a record structure of type `TaskThread`. The serialized iTask combinator is stored in the field `thrCallback`. In order to reconstruct the `*TSt` value on which the combinator function has to be applied, we also store the `TParams` information, which contains the appropriate `options`, `userId`, and `tasknr`. There is no need to store the `activated` token nor the accumulated `html` because this information gets accumulated from nodes above the subtree.

```
mkTaskThread :: (Task a) → Task a
mkTaskThread task = storeAndEvalThread
where storeAndEvalThread tst=:{activated,params}
      = case findThreadInTable params.tasknr tst of
          (Nothing,tst)  = storeAndEvalThread (insertNewThread
                           { thrParams    = params
                           , thrCallback = serializeClean task
                           } tst)
          (Just thr,tst) = evalTaskThread thr tst
```

For every task annotated with `UseAjax`, `mkTaskThread` is called. It stores the corresponding task, a state transition function, in the table if this has not been done in a previous incarnation (note that also this code might be re-evaluated several times). Finally it evaluates the task thread by calling `evalTaskThread`.

```
evalTaskThread :: TaskThread → Task a                                  1.
evalTaskThread {thrParams,thrCallback} = evalTask                      2.
where                                                                  3.
  evalTask tst=:{params,html}                                          4.
  # (a,tst=:{activated,html=nhtml})                                    5.
              = deserializeClean thrCallback                           6.
                {tst & params = thrParams, html = noHtml}              7.
  | activated                                                          8.
    = (a,{deleteThreads thrTaskNr tst & params = params})              9.
  | otherwise                                                          10.
    # newhtml = DivCode (showTaskNr thrParams.taskNr) nhtml            11.
    = (a,{tst & params = params, html = html +|+ newhtml})             12.
```

The function `evalTaskThread` can reconstruct the desired sub tree of the Task Tree. It is crucial to observe that this function can be called in any context. Therefore we can use it to regenerate the subtree when an Ajax call is done. In that case one first has to determine which thread from the `ThreadTable` should be selected. This is explained in Sect. 5.2.

The function `evalTaskThread` deserializes the stored iTask combinator and reconstructs the iTask `TSt` state such that the proper sub tree is reconstructed (lines 6-7). When the combinator task is finished (lines 8-9) the thread removes itself from the thread table. If the thread is not finished, more work on it has to be done in the future. The new Html code generated by the thread, `nhtml`, is appended to the HTML accumulator `html` marked by an Html `Div` construct that is labeled with the task number of the thread. This enables the JavaScript callback function on the client side to replace the old HTML code (which is labeled with the same task number) with the new one, leaving all other code unchanged. Therefore the part of the page that is updated depends on the chosen thread.

5.2 Determining which Threads to Activate Given an Event

Using callback functions for handling events is a common technique. However, in this case we cannot assign callback functions for an event beforehand because we deal with a distributed multi-user web enabled system. Due to global effects, a once constructed sub tree might not even exist anymore. As a matter of fact, when a task is no longer needed, all its administration is removed. Also its threads are removed from the thread table. We have to determine dynamically which thread is able to handle an event, if any. How can we do this?

The form committed by a user has been created by an iTask editor. Each iTask has a unique number, so we can encode this number in the event. The numbering discipline (Sect. 4.2) allows us to determine which thread to activate.

Assume that no global effects occurred. In the thread table we search for the ancestor thread that is closest related to the event: a task with the same prefix number. If such a thread can be found, and the corresponding task is indeed assigned to this particular user, it is evaluated. The subtree is reconstructed as described above and this subtree includes the basic iTask corresponding to the event. This task can handle the event as usual. If afterwards the chosen thread task is not finished yet, the corresponding Html code is communicated to the client (5.1) where the JavaScript callback function uses it to update the corresponding area on the web page. If the thread is finished, it removes itself from the table (5.1). Termination of this thread can trigger the evaluation of the next thread in the workflow structure. We search again in the thread table to find an enclosed thread which is now most closely related to the event and activate it. This process can repeat itself several times. Eventually, the page area that gets updated depends on the last thread activated in this way.

Assume that global effects did occur. How can we find out what has happened? We can find it out by reconstructing the whole Task Tree because this gives us the exact status of all worker tasks, but that is exactly what we wanted to avoid in the first place. Instead, for every worker we maintain an administration of type `GlobalEffect`, to keep track of global effects.

```
:: GlobalEffect = { versionNr      :: Int
                  , newThread      :: Bool
                  , deletedThreads :: [TaskNr] }
```

If a thread has become obsolete due to an action of another worker, its task number is added to the administration (`deletedThreads`) of the

worker of that task. If a new task is assigned to a specific worker, this fact is administrated in `newThread` as well. Also a version number is administrated for proper handling of browser buttons and cloning of windows.

The `GlobalEffect` administration is inspected before threads are determined. If there is a new thread, or if a thread has been deleted related to the event, we fall back to the old fashioned way and reconstruct the Task Tree starting from the root and construct a whole new page. Otherwise we can start looking for the right thread as described earlier. As a result, one cannot predict which part of a page will be updated. It can vary from a small area exactly covered by the closest thread, or a bigger area, or ultimately even the entire page.

5.3 Discussion

In this section we have shown how to incorporate Ajax threads in the iTask toolkit. As we have explained in Sect. 3, we have chosen to turn every worker task (i.e. a task assigned to a worker with the `@:` function) into an Ajax thread. As a result, the page that is displayed to a worker consists of a set of tasks, each of which can be handled individually by the worker without the need to wait for the full page to reload. The latter is only necessary in case her action has caused a non-local effect. In this way, the user experiences a smoothly operating workflow application. The workflow engineer can further fine-tune the workflow application by adding `UseAjax` annotations in the right places.

6. Implementing Local Task Rewriting on the Client

The contributions to the iTask toolkit described so far still result in a *thin-client* architecture: web browsers are used for rendering purposes, and all computations take place on the server. Any iTask that does not require server side database or file access can in principle be evaluated on the client instead of on the server. In this section we describe how this can incorporated within the iTask toolkit. Because task expressions are full-fledged Clean functions, and the iTask toolkit is based on generics, this means that non-trivial Clean code needs to run in a browser, which is something new. In Sect. 6.1 we show how we have done this by compiling an iTask program to two images. One image runs on the server and is a Clean executable, and one image runs as an *interpreted program* on every client. The two images run the same program, and reconstruct the Task Tree as described in the previous sections. Hence, also for the interpreted image callback functions need to be created and stored. This is described in Sect. 6.2. Again, non-local effects need to be taken into account. This is explained in Sect. 6.3. Finally, we discuss the achievements in Sect. 6.4.

6.1 Client Side Evaluation of Clean Code

The clients need to execute iTask expressions, which can be arbitrarily complex Clean expressions. One may choose to create a *plug-in* for Clean applications for these client browsers, but this conflicts with our design decision to implement iTasks as much as possible on existing web technology. Instead, we have chosen to make use of the Sapl interpreter [9]. Sapl is a very simple functional language in which only functions appear: it has no data structures, and no pattern matching. Due to its simplicity, the Sapl interpreter is very small. Despite its simplicity, it is much faster than interpreters like GHCi, Helium, Amanda and Hugs, albeit not as fast as the code generated by the Clean or GHC compiler (for more details see [9]). Being small and relatively fast it is a suited candidate to incorporate in a browser as a Java applet.

In order to make the Sapl interpreter suitable for the web, it had to be re-implemented in Java (the original version was encoded in C). Another crucial step is to create a compiler from Clean to Sapl. This has been done, and we present the details of this work elsewhere. The result is that we can compile a complete Clean iTask application to Sapl.

Every iTask application is now compiled to two images: one compiled by the Clean system to native Intel code running on the server, one on the client which is interpreted by Sapl. The advantage of this approach is that we obtain two, almost identical, images of the *same* iTask application between which we can switch. The code generated for the client differs slightly from the code generated for the server: the client cannot deal with global effects and will act differently in these situations. This happens under the hood: the workflow engineer is not concerned with these aspects. The two images are generated from one and the same iTask specification. The Sapl interpreter and Sapl code are loaded by the browser once when a worker visits the workflow page for the first time. In addition, one *single, generic* JavaScript is loaded as well that handles *all* Ajax communication. This overhead cost is paid only once.

6.2 Thread Storage and Creation

Now, whenever an `OnClient` annotation is specified for a task running either on the server or on the client, a modified `mkTaskThread` (Sect. 5.1) is called. The difference is that, when the `OnClient` annotation is encountered, not only a serialized version of the thread is made which can be executed on the server, but now also a serialized version of the thread is made which can be executed on the client. These two encodings are completely different though, due to the fact that we are dealing with two completely different implementations. So we need special conversion functions in Clean (and in Sapl) which can create the required serialization for Sapl (`serializeSapl` and `deserializeSapl`). To store the serialized client thread the thread table is extended with the field `thrCallbackClient`. The thread table is stored on the server as part of the state, and a copy of the relevant information of the thread table (only the threads intended for the particular client) is stored on the client as well.

```
serializeSapl   :: (Task a) → SaplSerialization
deserializeSapl :: SaplSerialization → Task a

:: TaskThread
  = { ... thrCallbackClient :: SaplSerialization ... }
```

If a client thread is created, the function `storeAndEvalThread` in `mkTaskThread` now additionally stores the serialized client thread for handling in Sapl. Furthermore, it sets the `tasklife` field in the `options` of the `*TSt` task state to the option `Client`. The `tasklife` field indicates where task information should be stored. The administration of the iTasks that should run on the client are stored in the browser page on the client. The setting propagates via this state to all subtasks being created by the thread. Hence for all subtasks it can be determined whether they should preferably run on the client or not. In the generated Html forms this knowledge is used in the encoding of the events.

6.3 Determining which Threads to Activate Given an Event

Initially, the evaluation of an iTask application starts on the server. It generates the first page containing the initial forms. The Sapl interpreter and Sapl code are loaded as a side-effect. When an event is generated, it is inspected on the client. There is one single special JavaScript script running for this purpose on the client side. This script operates as a switch: if the event is intended for the client, the script sends the event to the Sapl interpreter running as a Java applet. Otherwise the script sends the event to the server as if an ordinary `UseAjax` annotation was encountered.

The client basically performs the same actions as the server. However, the client cannot deal with global changes, or persistent storage handling on the server (e.g. database access). The general recipe is: in case of panic stop the execution on the client and fall back to the server side handling of the event.

There are two types of global effects: effects caused by the client that have an effect on co-workers. The client can recognize this situation and take the panic exit. Vice versa, co-workers can also cause a global effect that affect the client who is ignorant of these facts, for instance when it has not connected to the server for a long time. To catch this situation each client periodically has to ask the server whether global effects are stored for him in the `GlobalEffect` record (Sect. 5.2). The Ajax callback technology is ideal for handling this situation. In the case of global effects a boolean value can be set in the client such that the next event will be forced to communicate with the server such that client and server administration can be synchronized.

6.4 Discussion

With the `OnClient` annotation workflow engineers can create workflow systems that are able to compute arbitrarily complex computations on clients. This can significantly reduce the traditional round-trip communication between clients and server, it will reduce the workload of server applications, and it will enhance the worker experience as the workflow can respond quicker than in the old setting. Every workflow system has to be aware of non-local effects in any kind of implementation, and the iTask toolkit automatically detects whether this is the case and global Task Tree rewriting is required. The workflow engineer immediately profits from this approach because she does not need to concern herself with the question what parts of the local computations must take place on the client and what parts must be done on the server. The iTask toolkit automatically switches to global Task Tree rewriting in case a client task performs such work.

7. Related Work

The new iTask toolkit as described in this paper allows high level specification of multi-user workflows. Forms are generated generically from type information, which considerably decreases the amount of HTML programming. Complex dynamic workflows can be created that can be evaluated at both the client and the server without any restriction. Actions of workers can safely affect that of co-workers: the toolkit supports multi-user programming smoothly. The system is robust: computations that cannot be evaluated at the client side can always, and safely, be evaluated at the server side. We are not aware of any other functional system that has these features. However, there are functional approaches for handling web pages.

Links [4] and its recent extension formlets [5] is a functional language based web programming language. Links compiles to JavaScript for rendering HTML pages, and SQL to communicate with a back-end database. A Links program stores its session state at the client side. In a Links program, the keywords `client` and `server` force a top-level function to be executed at the client or server respectively. In Links processes can be spawned, and these processes can communicate via message passing. Client-server communication is implemented using Ajax technology. In iTasks processes are not created explicitly as in Links programs. The novel `UseAjax` and `OnClient` are similar to Links functionality, except that we do not limit their use to top-level functions, but instead allow any (nested) task to be annotated. In the iData and iTasks toolkits, forms are generated generically for every data type, whereas in Links and formlets these need to be coded by the programmer. Links and formlets are designed for form based web applications, as is the iData toolkit, whereas the iTask toolkit extends this with multi-user workflow, including recursive, higher-order workflows.

Another functional language based web programming language is Hop [13, 10]. Just as Links, Hop is compiled to JavaScript. It implements a strict separation between programming the user interface and the logic of an application. The main computation runs on the server, and the GUI runs on the client(s). These components can invoke each other (from GUI client to server via function calls, from server to client via signalling events). In particular the latter feature increases the expressive power of web applications, which are usually driven by the browser client side. Hop is a stratified language: each component is programmed in either one stratum in order to prevent it from performing operations that are considered to be illegal (e.g. database access by the GUI client, or rendering operations by the server application). Annotations control what stratum is used: within the main stratum (the server application code) ~ escapes to the GUI stratum, and within the GUI stratum one uses $ to escape to the server. Additional server logic can be invoked as a Hop service, which makes the design very modular. This has been implemented with Ajax. In the iData and iTasks toolkits, we do not require a stratified language approach to divide our attention to GUI programming versus application logic, because the GUI is mainly generated generically. The novel iTask annotations `UseAjax` and `OnClient` also allow fine-grained tuning of application logic, whereas an "exit strategy" is always available by relying on *Global Task Rewriting* strategy.

The Flapjax language [1] is an implementation of functional reactive programming in JavaScript. Many of its features are comparable with those of Hop, and indeed both are designed to create intricate web applications. The main difference with our approach is that the iTask system is geared for distributed, multi-user, workflow systems in which the coordination and interaction of work is defined in a highly declarative style.

The enabling technology of client-side evaluation in the iTask toolkit is Sapl. We have seen that the complete Clean application is compiled to Sapl. This enables our approach to use the full expressive power of Clean to perform intricate computations at the client side. A much more restricted approach has been implemented in Curry [7]: only a very restricted subset of Curry is translated to JavaScript to handle client side verification code fragments only.

8. Conclusions

In this paper we have presented a number of contributions to the iTask toolkit, a combinator library written in Clean to create workflow systems that run on the web in a pure functional style. The contributions to the workflow engineer are that she can annotate arbitrary task structures with two annotations: `UseAjax` and `OnClient`. Task structures annotated with `UseAjax` can be handled much more efficiently by the toolkit by using the underlying Ajax technology. Using this Ajax technology only the part of the web page corresponding to the task that is changed is updated instead of the entire web page. Task structures annotated with `OnClient` can be evaluated completely on the client side. This requires the Ajax technology since the client side evaluation of a single task only updates that task and hence only a fragment of the web page.

The actual gain in efficiency cannot be predicted on forehand because it highly depends on the kind of workflow being specified. The standard evaluation strategy of iTasks is already reasonably efficient thanks to global task rewriting. Local task rewriting is more efficient when there are no global effects. Otherwise local task rewriting will not be more efficient, the use of Ajax even introduces some minor additional run-time overhead. Local task rewriting will be more efficient than global task rewriting when the workflow is large, there are many users, and global effects occur occasionally.

Client side evaluation by Sapl has as advantage that internet traffic is avoided and that server processing load is relieved, but as disadvantage that the Sapl interpreter is slower than the compiled code generated by the Clean compiler. For arithmetical operations Clean can be an order of magnitude faster, but when higher order functions are being calculated Sapl performs quite well. Whether it is better to calculate on the client therefore depends on the size of the task, the kind of computations being performed, the amount of traffic on the internet, the speed of the network, the speed of the server, and the speed of the client. For small tasks the overhead of interpretation on the client usually outweighs the communication delay, even on slow client machines with fast internet connections. It is up to the workflow engineer to make the optimal choices. In any case, the advantage of both the `UseAjax` and `OnClient` annotation is the elimination of blank browser windows when waiting for a new page.

The technical contributions are the incorporation of Ajax technology within the iTask toolkit, the ability to convert any Clean expression to a Sapl function call, the introduction of Task Tree rewriting strategies that can automatically switch when required by the tasks that they evaluate, and rearranging the architecture of the iTask toolkit to incorporate these changes. We have maintained the declarative approach of the iTask toolkit. Everything is generated from an annotated, single source specification with a low burden on the workflow designer because the system itself switches automatically between client and server side evaluation when this is necessary without any effort of the workflow engineer. The iTask system integrates all mentioned technologies in a truly transparent and declarative way.

Acknowledgments

The authors would like to thank the anonymous reviewers for their constructive comments.

References

[1] The Flapjax site. http://www.flapjax-lang.org/.

[2] P. Achten. Clean for Haskell98 Programmers – A Quick Reference Guide –. Available at: `http://www.st.cs.ru.nl/papers/-2007/CleanHaskellQuickGuide.pdf`, July 13 2007.

[3] A. Alimarine and R. Plasmeijer. A Generic Programming Extension for Clean. In T. Arts and M. Mohnen, editors, *The 13th International workshop on the Implementation of Functional Languages, IFL'01, Selected Papers*, volume 2312 of *LNCS*, pages 168–186. Älvsjö, Sweden, Springer Verlag, Sept. 2002.

[4] E. Cooper, S. Lindley, P. Wadler, and J. Yallop. Links: Web programming without tiers. In *Proceedings of the 5th International Symposium on Formal Methods for Components and Objects (FMCO'06)*, volume 4709, CWI, Amsterdam, The Netherlands, 7 - 10 November 2006. Springer-Verlag.

[5] E. Cooper, S. Lindley, P. Wadler, and J. Yallop. An idiom's guide to formlets – draft –. Technical report, The University of Edinburgh, 2007. http://groups.inf.ed.ac.uk/links/papers/formlets-draft2007.pdf.

[6] J. Garrett. Ajax: A New Approach to Web Applications, 18 February 2005.

[7] M. Hanus. Putting Declarative Programming into the Web: Translating Curry to JavaScript. In *Proc. of the 9th International ACM SIGPLAN Conference on Principle and Practice of Declarative Programming (PPDP'07)*, pages 155–166. ACM Press, 2007.

[8] R. Hinze. A new approach to generic functional programming. In *The 27th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 119–132. Boston, Massachusetts, January 2000.

[9] J. Jansen, P. Koopman, and R. Plasmeijer. Efficient Interpretation by Transforming Data Types and Patterns to Functions. In H. Nilsson, editor, *Proceedings Seventh Symposium on Trends in Functional Programming, TFP 2006*, volume 7, pages 73–90, Nottingham, UK, The University of Nottingham, April 19-21 2006. Intellect Books.

[10] F. Loitsch and M. Serrano. Hop Client-Side Compilation. In *Seventh Symposium on Trends in Functional Programming, TFP 2007*, City College, New York, April 2-4 2007.

[11] R. Plasmeijer and P. Achten. The Implementation of iData - A Case Study in Generic Programming. In Andrew Butterfield and Clemens Grelck and Frank Huch, editor, *Implementation and Application of Functional Languages, 17th International Workshop, IFL 2005, Dublin, Ireland, September 19-21, 2005, Revised Selected Papers*, LNCS 4015, pages 106–123, Department of Computer Science, Trinity College, University of Dublin, September 19-21 2006.

[12] R. Plasmeijer, P. Achten, and P. Koopman. iTasks: Executable Specifications of Interactive Work Flow Systems for the Web. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming (ICFP 2007)*, pages 141–152, Freiburg, Germany, Oct 1–3 2007. ACM.

[13] M. Serrano, E. Gallesio, and F. Loitsch. Hop, a language for programming the web 2.0. In *Proceedings ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2006)*, pages 975 – 985, Portland, Oregon, USA, October 22-26 2006.

[14] W. van der Aalst, A. ter Hofstede, B. Kiepuszewski, and A. Barros. Workflow patterns. QUT Technical report, FIT-TR-2002-02, Queensland University of Technology, Brisbane, 2002.

[15] A. van Weelden. *Putting Types To Good Use*. PhD thesis, University of Nijmegen, The Netherlands, October 17 2007. ISBN 978-90-9022041-3.